Reduzierung der Programmgröße mit Hilfe von Klonerkennung

Bernhard J. Berger
Axivion GmbH
berger@axivion.com

Rainer Koschke
Arbeitsgruppe Softwaretechnik, FB 3 Informatik, Universität Bremen
koschke@informatik.uni-bremen.de

Abstract: In diesem Paper wird die Untersuchung beschrieben, ob sich mit Hilfe von einfachen Refactorings Klone aus einem bestehenden System in der Sprache C so entfernen lassen, dass die für Software im Automobilumfeld geltenden nichtfunktionalen Anforderungen nicht negativ beeinflusst werden. Motivation hierfür ist die Tatsache, dass kopierter Quelltext nicht nur die Wartung erschwert, sondern auch zur Folge hat, dass bestimmte Funktionalität mehrfach im Programm auftaucht und damit unnötig Speicherressourcen verbraucht.

1 Einleitung

Geklonter Code, also kopierter Quelltext, ist auf Platz eins von Martin Fowlers so genannter „Stinky Parade of Bad Smells“ [Fow00]. Diese Quelltextkopien legen die Vermutung nahe, dass sich hier ein Einsparpotenzial bietet, da diese Bereiche redundant sind und damit kompakter dargestellt werden können. In mehreren Untersuchungen wurde analysiert, wieviel Quelltext in einem Programm geklont wurde. Die Ergebnisse reichen im Normalfall von 7% bis 23% [Bak95, KDB+95, LPM+97] dupliziertem Code in einem Programm. In einem extremen Fall liegt sie sogar bei 59% kopiertem Quelltext [DRD99].

Software und die Softwareentwicklung nehmen in der Automobilbranche eine immer wichtigere Rolle ein, da sich in Kraftfahrzeugen immer mehr eingebettete Systeme befinden, auf denen immer größere Softwaresysteme laufen. Mit zunehmender Funktionalität, die ein einzelnes Steuergerät erfüllen muss, wird auch die Software größer und komplexer. Zugleich dürfen die nichtfunktionalen Anforderungen, die an die Software gestellt werden, wie Laufzeit und Speicherverbrauch, nicht negativ beeinflusst werden, damit der zur Verfügung stehende Mikrocontroller weiterverwendet werden kann und keine Fehler im Produkt auftreten.

Bei Klonen wird zwischen verschiedenen Klontypen [BKA+07] unterschieden, die ausdrücken, wie ähnlich sich die Quelltextabschnitte sind. Die Typ-1 Klone sind exakte Kopien, Typ-2 Klone enthalten Umbenennungen von Bezeichnern und Typ-3 Klone beinhalten zudem Einfügungen oder Auslassungen von Quellcode. Diese Art von Quelltextwieder-

verwendung hat einige nachteilige Auswirkungen, wie längerer Programmcode, duplizierte Fehler, sobald das Original fehlerhaft ist, aber auch ein größeres Programm. Dies führt langfristig zu einer schwerer zu verstehenden und zu wartenden Software, die immer fehleranfälliger wird. Eine höhere Fehlerquote führt langfristig zu höheren Ausgaben im Bereich des Softwaretests oder zur Beschädigung des Firmenimages, wenn die Fehler erst nach der Auslieferung auftreten.

Diese doch recht hohen Anteile an geklontem Quelltext werfen die Frage auf, ob es möglich ist, mit Hilfe geeigneter Methoden die Duplikate zusammenzufassen, um so den Quelltextumfang zu reduzieren. Sollte dies in größerem Stil möglich sein, so müssten sich hier positive Auswirkungen sowohl auf die Verständlichkeit des Quelltextes als auch den Speicherverbrauch des Programms im ROM des Steuergeräts zeigen. Letzteres hätte den positiven Nebeneffekt, dass ein bestehendes System mit weniger ROM für das Programm auskommen würde, und damit möglicherweise auf einem billigeren Mikrocontroller laufen könnte. Bei hohen Stückzahlen würden dadurch Kosten eingespart. Hierzu ist ein Verfahren wünschenswert, mit dem man mittels einer Reihe von Kennzahlen systematisch identifizieren kann, wann ein Klon ohne negative Beeinflussung der genannten nichtfunktionalen Anforderungen entfernt werden kann. Das vorgestellte Verfahren ist nicht auf das untersuchte System festgelegt und kann auch außerhalb der Automobilbranche eingesetzt werden.

2 Vorgehensweise

Im Rahmen der verwendeten Axivion-Bauhaus-Suite[1] stehen mehrere Klonerkennungstools zur Verfügung, die verschiedene Ansätze verfolgen. Es existieren tokenbasierte, syntaxbasierte und kombinierte Verfahren [KFF06, FKF08]. Für die tokenbasierten Techniken spricht die hohe Effizienz, für die syntaktischen ihre Eigenschaft, ausschließlich syntaktisch abgeschlossene Klone zu liefern. Das kombinierte Verfahren serialisiert den Syntaxbaum als Tokenstrom und kombiniert somit die hohe Geschwindigkeit der tokenbasierten Verfahren mit der höheren Präzision der syntaktischen Verfahren.

Die Qualität der Analyseergebnisse sowie die Vor- und Nachteile der verschiedenen Verfahren wurden bereits evaluiert [BKA+07, KFF06]. Vor dem Hintergrund des gewählten Ziels ergeben sich jedoch darüber hinaus weitere Gütekriterien für eine Klonerkennung. Da die Klone – nachdem sie gefunden wurden – entfernt werden sollen, ist ihre syntaktische Abgeschlossenheit ein sehr wichtiger Aspekt, der nur bei syntaxbasierten Analysen garantiert werden kann. Da viele Systeme in der Programmiersprache C geschrieben sind, sollte die Klonerkennung Präprozessoranweisungen berücksichtigen, was bei den aktuellen Analysen jedoch nur bei tokenbasierten Verfahren passiert. Somit sind diese beiden Punkte gegeneinander abzuwägen und ein geeignetes Verfahren zu wählen. Im Rahmen der durchgeführten Fallstudie wurde ein syntaxbasiertes Erkennungsverfahren ausgewählt, da dieses auf der bauhauseigenen Zwischendarstellung für Analysen arbeitet, auf der im weiteren Verlauf zusätzliche Untersuchungen durchgeführt werden können.

[1] `http://www.axivion.com`

Das die Präprozessoranweisungen und die dadurch entfernten Bereiche des Programmtextes nicht berücksichtigt werden, hat keine negative Auswirkung auf die gesetzten Ziele. Lässt man die Analyse für eine Präprozessorkonfiguration laufen, so sieht diese alle für diese Variante verwendeten Quelltextbereiche, wie sie für die eigentliche Übersetzung verwendet werden. Somit sind alle Teile, die Einfluss auf die genannten Anforderungen haben berücksichtigt. Ein Nachteil ist jedoch, dass die Analyse für jede verwendete Präprozessorkonfiguration laufen müsste, damit alle Klone gefunden würden. Gegen den Einsatz der kombinierenden Verfahren spricht leider, dass sich ihre Ergebnisse nicht auf die Zwischendarstellung beziehen, sondern auf den Quelltext. Hier wäre eine aufwändige Abbildung von den Ergebnissen auf die Zwischendarstellung notwendig. Diese Abbildung ist jedoch nicht in allen Fällen durchzuführen und so würden Teile der Ergebnisse nicht berücksichtigt werden.

Die gefundenen Klone sollten zunächst manuell auf Muster untersucht werden, um uninteressante Muster zu identifizieren, damit die Ergebnismenge in späteren Analyseläufen automatisiert verkleinert werden kann. Für die noch verbliebene Klonmenge wurden in einem anschließenden Schritt verschiedene Informationen erhoben. Hierzu gehörten einfache Metriken wie *Lines of Code* und die *McCabe Complexity* aber auch komplexere Informationen wie Anzahl an verschiedenen Datentypen und die Anzahl an Parametern, die eine extrahierte Funktion benötigen würde. Gerade für die letzten beiden Informationen musste ein Syntaxbaum vorhanden sein, um geeignete Aussagen treffen zu können. Anschließend wurden eine Reihe von Klonen, die durch einen Menschen als vielversprechend aussehend eingestuft wurden, manuell entfernt und die Auswirkungen auf das daraus resultierende Programm gemessen. Dies waren die Änderungen der Programmlänge des C-Programms, des Stackverbrauchs und die Anzahl der Assembleranweisungen der betroffenen Funktionen, da diese Messwerte direkten Einfluss auf die oben genannten nichtfunktionalen Anforderungen haben sollten. In einem letzten Schritt sollte versucht werden, einen Zusammenhang der Quelltext- und Klonmetriken auf die Programmmetriken zu finden, die die nichtfunktionalen Anforderungen beeinflussen.

3 Durchführung

Bei dem in der Fallstudie untersuchten System handelt es sich um eine Variante einer Motorsteuerungssoftware der Robert Bosch GmbH, die in der Sprache C implementiert ist und insgesamt 340.000 Anweisungszeilen umfasst. Ein großer Teil der vorhandenen Codebasis ist von Hand implementiert worden, es existieren jedoch auch generierte Abschnitte. Die Implementierungssprache hat direkte Auswirkungen auf die möglichen Arten, wie die Klone entfernt werden können. So bietet sich hier lediglich das Extrahieren von Funktionen als Möglichkeit an, da es in C weder Templates noch Vererbung gibt. Auf Grund der fehlenden Polymorphie können Algorithmen immer nur für einen bestimmten Datentyp implementiert werden, was ebenfalls die Arten Klone zu entfernen einschränkt. Das System wurde ausgewählt, da bei einer Inspektion des Codes mehrere ähnliche Stellen aufgefallen sind und sich die Frage ergab, ob diese alle vorhanden sein müssen oder ob man Quelltext einsparen könnte.

Unter den anfänglich 6000 gefunden Klonpaaren ließen sich schnell drei Klassen von Klonen finden, die auf Grund ihrer Beschaffenheit uninteressant für das gewählte Ziel waren und deshalb bei der Analyse automatisch entfernt werden konnten. Dies waren Typ-2 Klone die lediglich Assembleranweisungen, eine Sequenz von Zuweisungen oder eine leere Funktion enthielten. Diese Klone lassen sich auch leicht an den, bei der Programmierung vorgegebenen, Richtlinien festmachen. Alle Assembleranweisungen sollen in eigenständige Funktionen ausgelagert werden, was dazu führt, dass der Aufbau dieser Funktionen sehr einheitlich ist. Sie unterscheiden sich lediglich in der Anzahl der Parameter, der Deklaration von lokalen Variablen und der Rückgabe eines Wertes. Dazwischen ist eine *inline*-Assembleranweisung, die die Assembleranweisungen als einen String enthält. Die eigentliche Logik der Funktion ist in dem „Assembler-String“ enthalten, der von der Klonanalyse jedoch nur als einfacher String gewertet wird, der sich in seinem Inhalt unterscheidet. Sequenzen von einfachen Zuweisungen treten in der Sprache C häufig beim kopieren von Strukturen oder beim setzen von mehreren Feldern einer Struktur auf. Die leeren Funktionen finden ihre Begründung in der Architektur des Systems, in der jede Aufgabe von einem Prozess erledigt wird, der eine Initialisierungsfunktion sowie eine Hauptroutine enthalten muss. Aus diesem Grund ist eine Entfernung dieser leeren Funktionen nicht möglich. Diese sind für ein Refactoring uninteressant, da es zwar Klone im Sinne der oben genannten Definition sind, aber durch einen Menschen nicht als solche gewertet werden würden und zudem nicht sinnvoll entfernt werden könnten.

Anschließend wurden die genannten Metriken (*McCabe*, *LOC* und *Number of Statements*) und Analysen (*Anzahl benötigter Parameter* und *Typengleichheit von Variablen*) auf der Zwischendarstellung berechnet. Gerade für die Analysen ist die Zwischendarstellung wichtig, da hier an den Parametern und Variablen deren Typ annotiert ist und deshalb einfach verglichen werden können. Dies wäre auf Basis der Ergebnisse eines tokenbasierten Verfahrens schwierig gewesen, da dies auf textueller Ebene schwer festzustellen ist. Auf Basis der Berechnungsergebnisse wurde die Menge der Klone noch weiter minimiert. Zunächst wurden alle Klone entfernt, die auf Grund von unterschiedlichen Typen nicht generalisiert werden konnten. Danach wurden alle Klone entfernt, die nach ihrer Entfernung mehr als neun Parameter benötigen. Hierfür gibt es zwei Gründe; zum einen die Lesbar- und damit die Verständlichkeit und zum anderen kann der gewählte Zielprozessor lediglich fünf Parameter *by value* und vier Parameter *by reference* mittels Register übergeben. Jeder weitere Parameter wird über den Stack übergeben, was erhebliche nachteilige Auswirkungen auf den Stackverbrauch und die Laufzeit hat. Die Entfernung der Klassen sowie der Klone, die den genannten Kriterien entsprechen, führte dazu, dass fast die Hälfte der gefundenen Klone automatisch als uninteressant eingestuft wurden. Von der verbleibenden Menge wurden die Klone zur Entfernung ausgewählt, die entweder besonders viele Zeilen Quellcode oder Knoten in der Zwischendarstellung umfassten. [2] Diese Klone wurden zum Abschluss manuell entfernt und die bereits genannten Werte für die weitere Auswertung erhoben.

[2] Die genannten Werte korrelieren in dem untersuchten System nicht miteinander, da sehr viele Kommentare vorhanden sind und so kein direkter Zusammenhang zwischen der Klonlänge und der Anzahl der Knoten im Syntaxbaum besteht.

4 Ergebnisse

Bei der Auswertung der Daten wurde sehr schnell ersichtlich, dass es nur wenige Klone gibt, bei denen sich die Entfernung nicht oder nur minimal negativ auswirkt. Insgesamt wurden drei Klone gefunden, bei denen das resultierende Programm nach dem Refactoring weniger Speicherplatz verbraucht hat. Die Einsparungen liegen hier zwischen 116 und 1200 Byte.

Auf Grund der geringen gewonnenen Datenmenge lassen sich keine Zusammenhänge zwischen den erhobenen Messwerten erkennen. Zudem wird dies dadurch erschwert, dass die Software mit einer hohen Optimierungsstufe durch den Compiler übersetzt wird. Ohne diese Optimierung lassen sich eher Einsparungen erzielen und Zusammenhänge erkennen, wobei diese Art der Übersetzung eher unüblich und daher uninteressant ist.

5 Bewertung

Aus der Untersuchung lässt sich entnehmen, dass sich in dem analysierten System keine Quelltextkopien entfernen lassen, ohne die geltenden nichtfunktionalen Anforderungen negativ zu beeinflussen. Daraus kann jedoch nicht gefolgert werden, dass in dem System keine Klone vorhanden sind, sondern sich diese nicht mit den Mitteln der Sprache C effektiv entfernen lassen. Es stellt sich natürlich die Frage, ob die gewählte syntaxbasierte Klonerkennungstechnik, hierfür am besten geeignet ist oder ob ein anderes Analyseverfahren, welches sich nicht auf die syntaktische Ähnlichkeit, sondern auf die semantische Ähnlichkeit stützt, geeigneter gewesen wäre.

So finden sich Dateien, die ein menschlicher Gutachter sofort als Klon identifiziert. Bei genauerer Betrachtung erkennt man jedoch, dass es sich hierbei um manuell instantiierte Templates handelt, die auf verschiedenen Typen operieren, was in C nicht kompakter ausgedrückt werden kann. Ein zweiter Klon zeigt eindeutig objektorientierte Strukturen. Es handelt sich um zwei Strukturen sowie eine Reihe von zugehörigen Funktionen. Die zweite Struktur enthält alle Elemente der ersten Struktur und hat einige weitere Felder. Die Funktionen auf diesen Strukturen sind gleich, bis auf die Stellen, wo bei der zweiten Struktur die zusätzlichen Elemente verwendet werden. Dies würde in einer objektorientierten Sprache als zwei Klassen mit einer Vererbungsbeziehung modelliert werden, was in C jedoch nicht möglich ist.

Diese Erkenntnisse legen nahe, dass das Thema Klonmanagement im Fall dieser Software wichtig ist, damit Änderungen nicht versehentlich inkonsistent durchgeführt werden und so Fehler im System entstehen. Zudem zeigt sich, dass ein solch komplexes System mittels einer Sprache, die objektorientierten Paradigmen bietet, redundanzfreier formuliert werden könnte. Schließlich hat sich das prinzipielle Vorgehen, über Kennzahlen die Menge der potentiell lohnenswerten Entfernungen automatisch einzuengen, bewährt, da nur sehr wenige Klone einer menschlichen Begutachtung unterzogen werden mussten.

Literatur

[Bak95] Brenda S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Working Conference on Reverse Engineering*. IEEE CS Press, 1995.

[BKA+07] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke und Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, September 2007.

[DRD99] Stéphane Ducasse, Matthias Rieger und Serge Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *ICSM*, 1999.

[FKF08] Raimar Falke, Rainer Koschke und Pierre Frenzel. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empiricial Software Engineering*, 2008. accepted for publication.

[Fow00] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[KDB+95] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler und Ettore Merlo. Pattern matching for design concept localization. In *Working Conference on Reverse Engineering*. IEEE Computer Society Press, 1995.

[KFF06] Rainer Koschke, Raimar Falke und Pierre Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Working Conference on Reverse Engineering*, Seiten 253–262. IEEE CS Press, 2006.

[LPM+97] B. Lague, D. Proulx, J. Mayrand, E.M. Merlo und J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *International Conference on Software Maintenance*, Seiten 314–321, 1997.